*PBHD: An Efficient Graph Representation for Floating Point Circuit Verification

Yirng-An Chen   Randal E. Bryant

May 1997
CMU-CS-97-134

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

*BMDs, HDDs, and K*BMDs provide compact representations for functions which map Boolean vectors into integer values, but not floating point values. In this paper, we propose a new data structure, called Multiplicative Power Binary Hybrid Diagrams (*PBHDs), to provide a compact representation for functions that map Boolean vectors into integer or floating point values. The size of the graph to represent the IEEE floating point encoding is linear with the word size. The complexity of floating point multiplication grows linearly with the word size. The complexity of floating point addition grows exponentially with the size of the exponent part, but linearly with the size of the mantissa part. We applied *PBHDs to verify integer multipliers and floating point multipliers before the rounding stage, based on a hierarchical verification approach. For integer multipliers, our results are at least 6 times faster than *BMD's. Previous attempts at verifying floating point multipliers required manual intervention. We verified floating point multipliers before the rounding stage automatically.

This research is sponsored by the Defense Advanced Research Projects Agency (DARPA) under contract number DABT63-96-C-0071.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. Goverment.

Keywords: Binary Moment Diagram, *BMD, Hybrid Decision Diagram, HDD, K*BMD, Multiplicative Power Binary Hybrid Diagram, *PBHD, Arithmetic Circuit, IEEE Floating Point Standard, Formal Verification

1 Introduction

Binary Moment Diagrams (BMDs) [3] have proved successful for representing and manipulating functions mapping Boolean vectors to integer values symbolically. They have been used in the verification of arithmetic circuits [4]. Clarke, *et al.* [6] extended BMDs to a form they call Hybrid Decision Diagrams (HDDs), where a function may be decomposed with respect to each variable in one of six ways, but without edge weights. Drechsler, *et al.* [9] extended Multiplicative BMDs (*BMDs) to a form called K*BMDs, where a function may be decomposed with respect to each variable in one of three ways, and with both additive and multiplicative edge weights. None of these diagrams can represent functions which map Boolean vectors to floating point values, unless rational numbers are introduced into the representation [2]. But using rational numbers in the representation requires more space to store the numerator and denominator separately, and more computation to extract the rational numbers.

Verification of floating point arithmetic circuits with any of these three diagrams requires floating point circuits to be divided into several sub-circuits, for which specifications can be expressed in terms of integer functions and their operations [4, 5, 7]. The correctness of the overall circuit must be proved by users from the specifications of the verified sub-circuits. For instance, the floating point multiplier was divided into the circuits for the mantissa multiplication, the exponent addition, and the rounding in [5]. The verification of these three sub-circuits was performed automatically by word-level SMV [7], but the correctness of the entire multiplier must be proved by users from the verified specifications of these three sub-circuits. To avoid partitioning floating point arithmetic circuits for verification, it is necessary to have decision diagrams that represent and manipulate the floating point functions efficiently.

In this paper, we propose a new representation, called Multiplicative Power Binary Hybrid Diagrams (*PBHDs), that improves on previous diagrams in representing floating point functions. *PBHDs can represent functions having Boolean variables as arguments and floating values as results. This structure is similar to that of HDDs [6], except that they are based on powers-of-2 edge weights and complement edges for negation. We show that the size of floating point multiplication grows linearly with the word size. For floating point addition, we show that the complexity grows exponentially with the exponent size, but linearly with the mantissa size.

Based on a hierarchical verification methodology [3, 4], we have applied *PBHDs to verify different sizes and types of integer multipliers. Compared with *BMDs, *PBHDs are consistently six times faster in terms of CPU time and use less memory. We have also applied *PBHDs to verify different sizes and types of floating point multipliers before the rounding stage, which had never been verified symbolically and automatically. Our results show that the verification of floating point multipliers requires minimal effort beyond integer multipliers. Our next step is to look into the rounding stage and entire floating point multipliers. Earlier results using HDDs [5] show that the rounding stage itself can be handled.

The outline for the rest of this paper is as follows: Section 2 discusses basic properties and limitations of BMDs, *BMDs and HDDs. The *PBHD data structure is presented in Section 3. Section 4 shows *PBHDs for several numeric functions including integers and floating point numbers. Section 5 compares the differences among *PBHD, *BMD, HDD and K*BMD. The performance results are shown in Section 6. Finally, Section 7 contains the conclusion and possible future work.

2 Basics and Limitations of BMDs, *BMDs and HDDs

For expressing functions Boolean variables into integer values, BMDs[3] use the moment decomposition of a function:

$$\begin{aligned} f &= (1-x)\cdot f_{\overline{x}} + x\cdot f_x \\ &= f_{\overline{x}} + x\cdot (f_x - f_{\overline{x}}) \\ &= f_{\overline{x}} + x\cdot f_{\delta x} \end{aligned} \tag{1}$$

where $\cdot$,+ and $-$ denote multiplication, addition and subtraction, respectively. Term f_x ($f_{\overline{x}}$) denotes the positive (negative) cofactor of f with respect to variable x, i.e., the function resulting when constant 1 (0) is substituted for x. By rearranging the terms, we obtained the third line of Equation 1. Here, $f_{\delta x} = f_x - f_{\overline{x}}$ is called the linear moment of f with respect to x. This terminology arises by viewing f as being a linear function with respect to its variables, and thus $f_{\delta x}$ is the partial derivative of f with respect to x. The negative cofactor $f_{\overline{x}}$ will be termed the constant moment, i.e., it denotes the portion of function f that remains constant with respect to x. This decomposition is also called Davio positive in K*BMDs [9]. Each vertex of a BMD describes a function in terms of its moment decomposition with respect to the variable labeling the vertex. The two outgoing arcs denote the constant and linear moments of the function with respect to the variable.

Clarke, *et al.* [6] extended BMDs to a form they call Hybrid Decision Diagrams (HDDs), where a function may be decomposed with respect to each variable in one of six decomposition types. In our experience with HDDs, we found that three of their six decomposition types are useful in the verification of arithmetic circuits. These three decomposition types are Shannon, Davio Positive, and Davio Negative. Therefore, Equation 1 is generalized to the following three equations according the variable's decomposition type:

$$f = \begin{cases} (1-x)\cdot f_{\overline{x}} + x\cdot f_x & (Shannon) \\ f_{\overline{x}} + x\cdot f_{\delta x} & (Davio\ Positive) \\ f_x + (1-x)\cdot f_{\delta \overline{x}} & (Davio\ Negative) \end{cases} \tag{2}$$

Here, $f_{\delta \overline{x}} = f_{\overline{x}} - f_x$ is the partial derivative of f with respect to $\overline{x}$. The BMD representation is a subset of HDDs. In other word, the HDD graph is the same as the BMD graph, if all of the variables use Davio positive decomposition.

As an example, Figure 1 show an integer function f with Boolean variables x and y that is represented by a truth table, BMDs, *BMDs, and HDDs with Shannon decompositions (also called MTBDD [8]). In our drawing, the variables with Shannon and Davio positive decomposition types are drawn in shadowed and non-shadowed vertices respectively. Figure 1.b shows the BMD representation. To construct this graph, we apply Equation 1 to function f recursively. First, with respect to variable x, we can get $f_{\overline{x}} = 1 + y$, represented as the graph of the dashed-edge of vertex x, and $f_{\delta x} = 3 + 3y$, represented by the solid branch of vertex x. Observe that $f_{\delta x}$ can be expressed by $3 \times f_{\overline{x}}$. By extracting the factor 3 from $f_{\delta x}$, the graph became Figure 1.c. This graph is called a Multiplicative BMD (*BMD) which extracts the greatest common divisor (GCD) from both

x	y	f
0	0	1
0	1	2
1	0	4
1	1	8

(a)

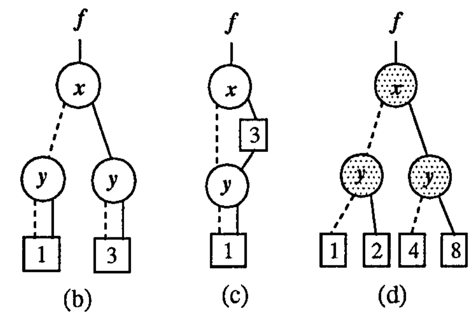


Figure 1: **An integer function with Boolean variables, $f = 1 + y + 3x + 3xy$, is represented by (a) Truth table, (b) BMDs, (c) *BMDs, (d) HDDs with Shannon decompositions.** The dashed-edges are 0-branches and the solid-edges are the 1-branches. The variables with Shannon and Davio positive decomposition types are drawn in shadowed and non-shadowed vertices respectively.

branches. The edge weights combine multiplicatively. The HDD with Shannon decompositions can be constructed from the truth table. The dashed branch of vertex x is constructed from the first two entries of the table, and the solid branch of vertex x is constructed from the last two entries of the table.

Observe that if variables x and y are viewed as bits forming 2-bit binary number, X=y+2x, then the function f can be rewritten as $f = 2^{(y+2x)} = 2^X$. Observe that HDDs with Shannon decompositions and BMDs grow exponentially for this type of functions. *BMDs can represent them efficiently, due to the edge weights. However, *BMDs cannot represent the functions as $f = 2^{X-B}$, where B is a constant, because it can only represent integer functions.

3 The *PBHD Data Structure

In this section, we introduce a new data structure, Multiplicative Power Binary Hybrid Diagrams (*PBHDs), to represent functions that map Boolean vectors to integer or floating point values. This structure is similar to that of HDDs, except that they use powers-of-2 edge weights and negation edges. The powers-of-2 edge weight feature enables us to represent and manipulate functions mapping Boolean vectors to floating point values. Negation edges can further reduce graph size by half. As with *PBHDs, we assume that there is a total ordering of the variables such that the variables are tested according to this ordering along any path from the root to a leaf. Each variable is associated with its own decomposition type and all nodes of that variable use the corresponding decomposition.

3.1 Edge Weights

*PBHDs use three of HDD's six decompositions expressed in Equation 2. Similar to *BMDs, we adapt the concept of edge weights to *PBHDs. Unlike *BMD's edge weights, we restrict our edge weights to be powers of a constant c. Thus, Equation 2 is rewritten as:

$$\langle w, f\rangle = \begin{cases} c^w \cdot (((1-x)\cdot f_{\bar{x}} + x \cdot f_x) & (Shannon) \\ c^w \cdot (f_{\bar{x}} + x \cdot f_{\delta x}) & (Davio\ Positive) \\ c^w \cdot (f_x + (1-x)\cdot f_{\delta \bar{x}}) & (Davio\ Negative) \end{cases}$$

where $\langle w, f\rangle$ denotes $c^w \times f$. In general, the constant c can be any positive integer. Since the base value of the exponent part of the IEEE floating point format is 2, we will consider only $c = 2$ for the remainder of the paper. Observe that w can be negative, i.e. we can represent rational numbers. The power edge weights enable us to represent functions mapping Boolean variables to floating point values without using rational numbers in our representation.

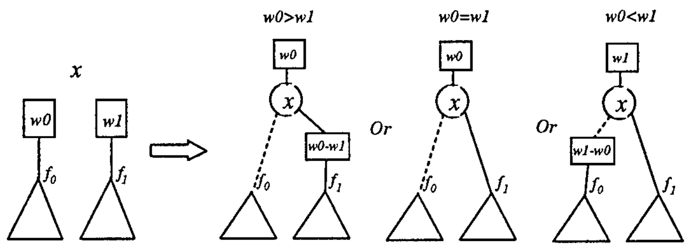


Figure 2: **Normalizing the edge weights.**

In addition to HDD's reduction rules [6], we apply several edge weight manipulating rules to maintain the canonical form of the resulting graph. Let $w0$ and $w1$ denote the weights at branch 0 and 1 respectively, and f_0 and f_1 denote the functions represented by branch 0 and 1. To normalize the edge weights, we chose to extract the minimum of the edge weight $w0$ and $w1$. This is a much simpler computation than the GCD of integer *BMDs or the reciprocal of rational *BMDs [2]. Figure 2 illustrates the manipulation of edge weights to maintain a canonical form. The first step is to extract the minimum of $w0$ and $w1$. Then, the new edge weights are adjusted by subtracting the minimum from $w0$ and $w1$ respectively. A node is created with the index of the variable, the new edge weights, f_0 and f_1. Base on the relation of $w0$ and $w1$, the resulting graph is one of three graphs in Figure 2. Note that at least one branch has zero weight. In addition, the manipulation rule of the edge weight is the same for all of the three decomposition types. In other words, the representation is normalized if and only if the following holds:

- The leaf nodes can only have odd integers or 0.
- At most one branch has non-zero weight.
- The edge weights are greater than or equal to 0, except the top one.

3.2 Negation Edge

Negation edges are commonly used in BDDs [1] and KFDDs [10], but not in *BMDs, HDDs and K*BMDs. Since our edge weights extract powers-of-2 which are always positive, negation edges are added to *PBHDs to increase sharing among the diagrams. In *PBHD, the negation edge of function f represents the negation of f. Note that $-f$ is different from $\overline{f}$ for Boolean functions.

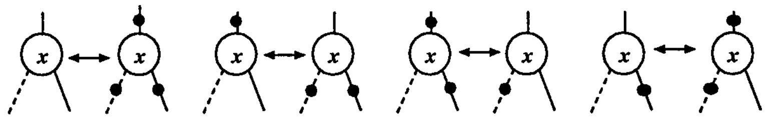


Figure 3: **Rules for negation edges**

Negation edges allow greater sharing and make negation a constant computation. In our *PBHD data structure, we use the low order bit the pointers to denote negation, as is done with the complement edge of BDDs. To maintain a canonical form, we must constrain the use of negation edges. Unlike KFDDs [10], where Shannon decompositions use a different method from Davio positive and negative decompositions, *PBHDs use the same method for manipulating the negation edge for all three decomposition types. The *PBHDs must follow these rules: the zero edge of every node must be a regular edge, the negation of leaf 0 is still leaf 0, and leaves must be nonnegative. Figure 3 illustrates the rules for negation edges. These four pair of functions are functionally equivalent. In our implementation, we always create nodes for the left side of the pair. In other words, the 0-branch is always positive. This guarantees a canonical form.

4 Representation of Numeric Functions

*PBHDs can effectively represent numeric functions that map Boolean vectors into integer or floating point values. We first show that *PBHDs can represent integer functions with comparable sizes to *BMDs. Then, we show the *PBHD representation for floating point functions and operations.

4.1 Representation of Integers

*PBHDs, similar to *BMDs, can provide a concise representation of functions which map Boolean vectors to integer values. Let $\vec{x}$ represent a vector of Boolean variables: $x_{n-1}, \ldots, x_1, x_0$. These variables can be considered to represent an integer X according to some encoding, e.g., unsigned binary or two's complement. Figure 4 illustrates the *PBHD representations of several common encodings for integers. In our drawing of *PBHDs, we indicate the edge weight and leaf node in shadowed and non-shadowed square boxes respectively. Unlabeled edges have weight 0 (2^0). The dashed line from a vertex denotes the case where the vertex variable is evaluated to 0, and the solid line denotes the case where the vertex variable is evaluated to 1. An unsigned number is encoded as a sum of weighted bits. The *PBHD representation has a simple linear structure where the leaf values are formed by the corresponding edge weight and leaf 1 or 0. For representing signed numbers, we

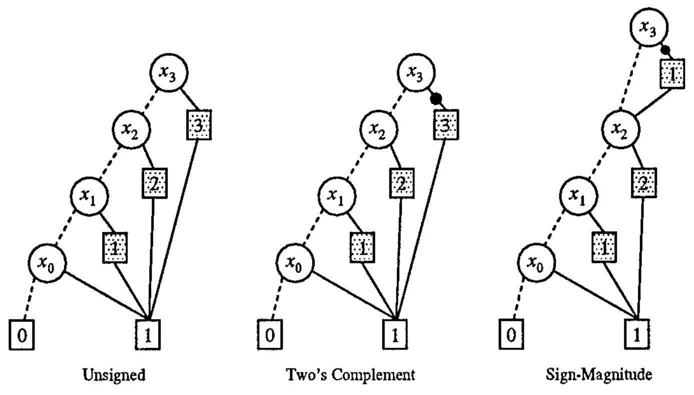


Figure 4: ***PBHD Representations of Integers.** Each variable uses Davio positive decomposition. All commonly used encodings can be represented with linear complexity.

assume x_{n-1} is the sign bit. The two's complement encoding has a *PBHD representation similar to that of unsigned integers, but with bit x_{n-1} having weight -2^{n-1} represented by the edge weight $n-1$ and the negation edge. Sign-magnitude integers also have *PBHD representations of linear complexity, but with the constant moment with respect to x_{n-1} scaling the remaining unsigned number by 1, and the linear moment scaling the number by -2 represented by edge weight 1 and the negation edge. In evaluating the function for $x_{n-1} = 1$, we would add these two moments, effectively scaling the number by -1. Note that it is more logical to use Shannon decomposition for the sign bit.

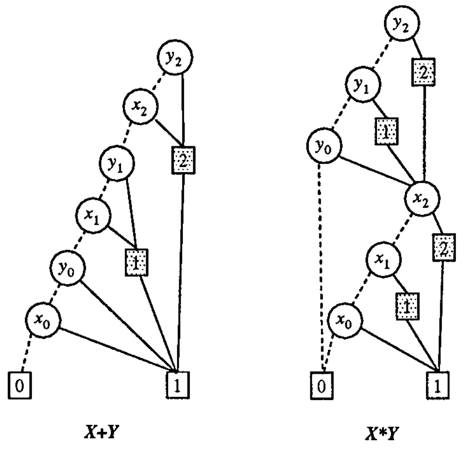


Figure 5: ***PBHD Representations of Word-Level Sum and Product.** Each variable uses Davio positive decomposition. The graphs grow linearly with word size.

Figure 5 illustrates the *PBHD representations of several common arithmetic operations on integer data. Observe that the sizes of the graphs grow only linearly with the word size n. Integer addition can be viewed as summing a set of weighted bits, where bits x_i and y_i both have weight 2^i represented by edge weight i. Integer multiplication can be viewed as summing a set of partial products of the form $x_i 2^i Y$. In summary, while representing the integer functions, *PBHDs with Davio positive decompositions usually will get the most compact representation among these three decompositions.

4.2 Representation of Floating Point Numbers

Let us consider the representation of floating point numbers by IEEE standard 754. For example, the double-precision numbers are stored in 64 bits: 1 bit for the sign (S_x), 11 bits for the exponent (EX), and 52 bits for the mantissa (X). The exponent is a signed number represented with bias (B) of 1023. The mantissa represents a number less than 1. Based on the value of the exponent, the IEEE floating point format can be divided into four cases:

$$\begin{cases} (-1)^{S_x} \times 1.X \times 2^{EX-B} & If\ 0 < EX < All\ 1 & (normal\ numbers) \\ (-1)^{S_x} \times 0.X \times 2^{1-B} & If\ EX = 0 & (denormal\ numbers) \\ NaN & If\ EX = All\ 1\ \&\ X \neq 0 & \\ (-1)^{S_x} \times \infty & If\ EX = All\ 1\ \&\ X = 0 & \end{cases}$$

Currently, *PBHDs cannot handle infinity and NaN (not a number) cases in the floating point representation. Instead, assume they are normal numbers.

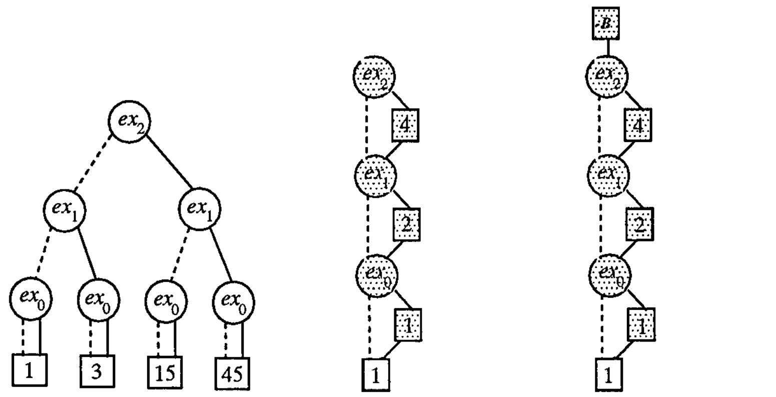


Figure 6: ***PBHD Representations of 2^{EX} and 2^{EX-B}.** The graph grows linearly in the word size with Shannon, but grows exponentially with Davio Positive.

Figure 6 shows *PBHD representations for 2^{EX} and 2^{EX-B} using different decompositions. To represent function c^{EX} (in this case $c = 2$), *PBHDs express the function as a product of factors of the form $c^{2^i ex_i} = (c^{2^i})^{ex_i}$. In the graph with Shannon decompositions, a vertex labeled by variable cx_i has outgoing edges with weights 0 and c^{2^i}, both leading to a common vertex denoting the product of the remaining factors. But in the graph with Davio positive decompositions, there is no sharing except for the vertices on the layer just above the leaf nodes. Observe that the size of *PBHDs with Davio positive decomposition grows exponentially in the word size while the size of *PBHDs with Shannon grows linearly. Interestingly, *BMDs have a linear growth for this type of function, while *PBHDs with Davio positive decompositions grow exponentially.

To represent floating point functions symbolically, it is necessary to represent 2^{EX-B} efficiently, where B is a constant. HDDs cannot represent both 2^{EX} and 2^{EX-B} efficiently, since they do not have edge weights. *BMDs can represent 2^{EX} efficiently [3], but not 2^{EX-B}, which can be interpreted as 2^{EX} divided by 2^{B}. K*BMDs have the same problem as *BMDs in representing 2^{EX}-B. Without using rational numbers, *BMDs, HDDs and K*BMDs cannot represent 2^{EX-B}. However, the edge weights in *PBHDs can represent function 2^{EX-B} efficiently by simply adding a edge weight $-B$ on top of the *PBHDs of 2^{EX} as shown in Figure 6.c.

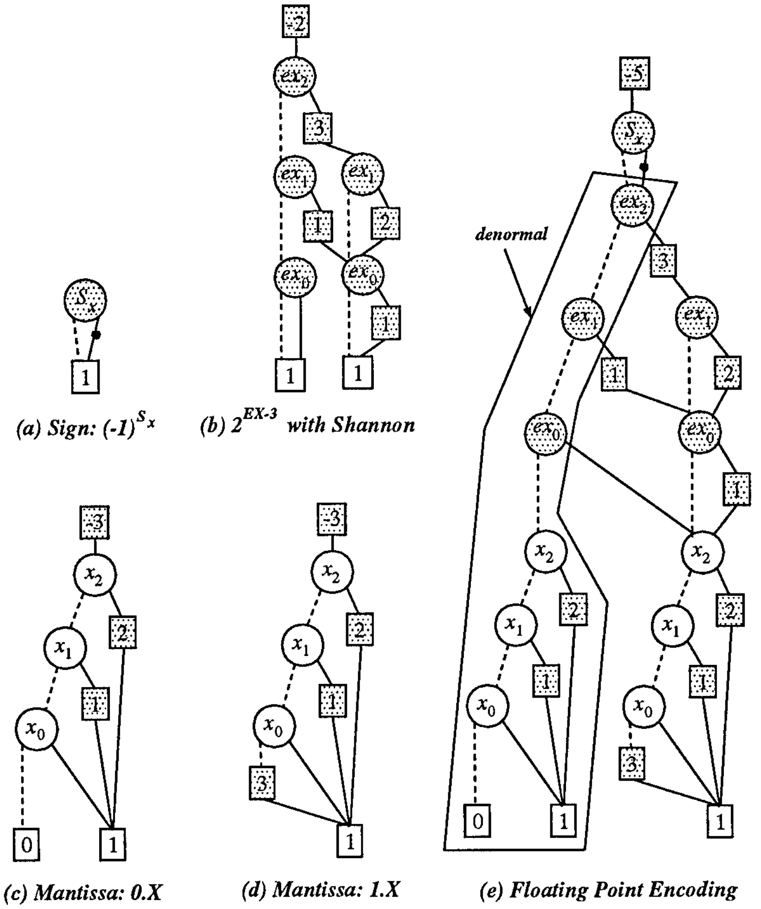


Figure 7: **Representations of the floating point encodings.**

Figure 7 shows the *PBHD representations for each component of the floating point encoding and the whole representation, where EX has 3 bits, X has 4 bits, and the bias B is 3. The sign S_x and $\vec{cx}$ variables use Shannon decomposition, while variables $\vec{x}$ use Davio positive. Figure 7.a shows the *PBHD representation for the sign bit $(-1)^{S_x}$. When S_x is 0, the value is 1; otherwise, the value is -1 represented by the negation edge and leaf node 1. Figure 7.b shows the *PBHD representation for the exponent part 2^{EX-3}. The graph is more complicated than Figure 6.c, because in the floating point encoding, when EX =0, the value of the exponent is $1 - B$ instead of $-B$. Observe that each exponent variable, except the top variable cx_2, has two nodes: one to represent the denormal number case and another to represent normal number case. Figure 7.c shows the representation for the mantissa part $0.X$ obtained by dividing X by 2^{-3}. Again, the division by powers of 2 is just adding the edge weight on top of the original graph. Figure 7.d shows the representation for the mantissa part $1.X$ which is the sum of $0.X$ and 1. The weight (2^{-3}) of the least significant bit is extracted to the top and the leading bit 1 is represented by the path with all variables set to 0. Finally, Figure 7.e shows the *PBHD representation for the floating point encoding. Observe that negation edges reduce the graph size by half. The polygon in the figure denotes the representation for denormal numbers. The rest of the graph represents the normal numbers. Assume the exponent is n bits and the mantissa is m bits. Note that the edge weights are encoded into the node structure in our implementation, but the top edge weight requires an extra node. It can be shown that the total number of *PBHD nodes for the floating point encoding is $2(n+m)+3$. Therefore, the size of the graph grows linearly with word size. In our experience, it is more logical to use Shannon decompositions for the sign and exponent bits, and Davio positive decompositions for the mantissa bits. For the rest of the paper, we use this decomposition setting to represent the floating point encoding.

4.3 Floating Point Multiplication and Addition

This section presents floating point multiplication and addition based on *PBHDs. Here, we show the representations of these operations before rounding. In other words, the resulting *PBHDs represent the precise results of the floating point operations. For floating point multiplication, the size of the resulting graph grows linearly with the word size. For floating point addition, the size of the resulting graph grows exponentially with the size of the exponent part.

Let $F_X = (-1)^{S_x} \times v_x.X \times 2^{EX-B}$ and $F_Y = (-1)^{S_y} \times v_y.X \times 2^{EY-B}$, where v_x (v_y) is 0 if EX (EY) = 0, otherwise, v_x (v_y) is 1. EX and EY are n bits, and X and Y are m bits. Let the variable ordering be the sign variables, followed by the exponent variables and then the mantissa variables. Based on the value of EX, Expanding and rearranging the terms of the multiplication $F_X \times F_Y$ yields:

$$\begin{aligned} F_X \times F_Y &= (-1)^{S_x \oplus S_y} \times (v_x.X \times 2^{EX-B}) \times (v_y.Y \times 2^{EY-B}) \\ &= (-1)^{S_x \oplus S_y} \times 2^{-2B} \times \begin{cases} 2^1 \times (0.X \times v_y.Y) \times 2^{EY} & Case\ 0: \ EX = 0 \\ 2^{EX} \times (1.X \times (v_y.Y) \times 2^{EY} & Case\ 1: \ EX \neq 0 \end{cases} \end{aligned} \quad (3)$$

Figure 8 illustrates the *PBHD representation for floating point multiplication. Observe that two negation edges reduce the graph size to a quarter of the original size. When $cx_1 = cx_0 = 0$(as Case 0 in Equation 3), the sub-graph represents the function $(0.X \times v_y.Y) \times 2^{EY}$. When $cx_1 \neq 0$

or $cx_0 \neq 0$ (Case 1), the sub-graph represents the function $(1.X \times v_y.Y) \times 2^{EY}$. Observe that there is no sharing in the exponent parts of these two sub-graphs, but there are some sharings in the graphs of the mantissa products, The lower part of the resulting graph shows four mantissa products(from left to right): $X \times Y$, $(2^3 + X) \times Y$, $X \times (2^3 + Y)$, $(2^3 + X) \times (2^3 + Y)$. The first and third mantissa products share the common sub-function Y shown by the solid rectangles in Figure 8. The second and fourth products share the common sub-function $2^3 + Y$ shown by the dashed rectangles in Figure 8. The following theorem shows that the size of the resulting graph grows linearly with the word size for the floating point multiplication.

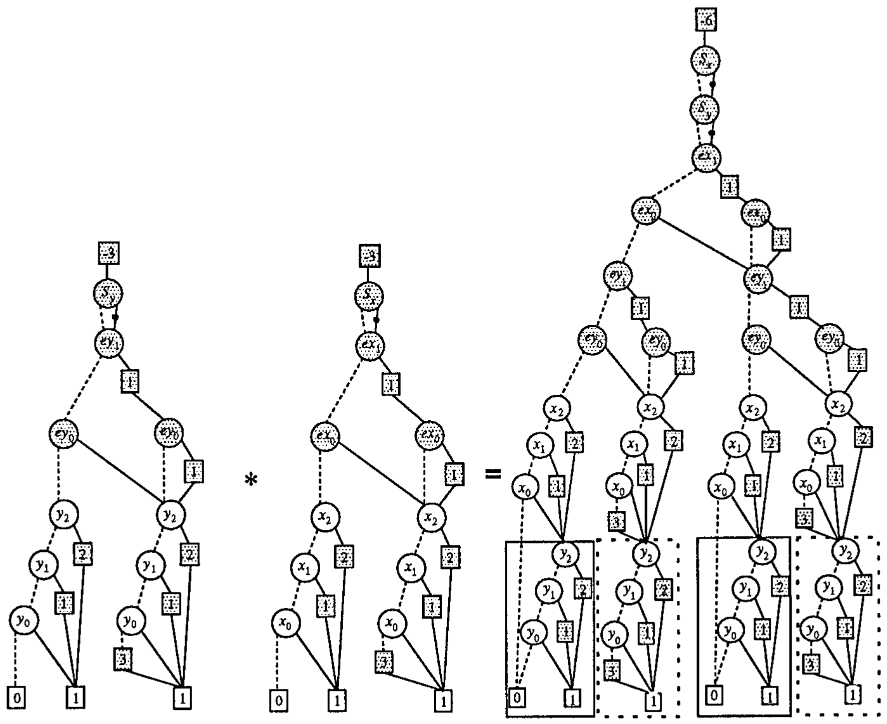


Figure 8: **Representations of the floating point multiplication.**

Theorem 1 *Given the ordering as shown in Figure 8, the size of the resulting graph of floating point multiplication is* $6(n + m) + 3$, *where* n *and* m *are the number of bits in the exponent and mantissa parts.*

Proof: From Equation 3 and Figure 8, we know that there are no sharing in the sub-graphs for $EX = 0$ and $EX \neq 0$. For $EX = 0$, the size of the sub-graph except leaf nodes is the sum of the nodes for the exponent of F_Y ($2n - 1$ nodes), the nodes for the mantissa of F_X ($2m$ nodes), and the nodes for the mantissa of F_Y (m nodes). Similarly, for $EX \neq 0$, the size of the sub-graph except leaf nodes is also $2n + 3m - 1$. The size for the exponent part of F_X is $2n - 1$. The number of nodes for the sign bits and top level edge weight is 3, and the number of leaf nodes is 2. Therefore, the size of the resulting graph for floating point multiplication is $6(n+m)+3$. □

For floating point addition, the size of the resulting graph grows exponentially with the size of the exponent part. The following theorem proves that the number of distinct mantissa sums in *PBHD representation grows exponentially with the size of the exponent part.

Theorem 2 *For floating point addition $F_X + F_Y$, the number of distinct mantissa sums is $2^{n+3} - 10$, where n is the number of bits in the exponent part.*

Proof: We first show that the floating point addition can be divided into two cases according to the sign bits of two operands. When $S_x \oplus S_y$ is equal to 0, the floating point addition must be performed as "true addition", shown as Equation 4.

$$F_X + F_Y \quad = \quad (-1)^{S_x} \times (2^{EX-B} \times v_x.X + v_y.Y \times 2^{EY-B}) \tag{4}$$

When $S_x \oplus S_y$ is equal to 1(i.e., they have different sign), the floating point addition must be performed as "true subtraction", shown as the following equation.

$$F_X + F_Y \quad = \quad (-1)^{S_x} \times (2^{EX-B} \times v_x.X - v_y.Y \times 2^{EY-B}) \tag{5}$$

There is common distinct mantissa sum among true addition and true subtraction, since one performs addition and another performs subtraction.

Let us consider the true addition operation first. Based on the relation of EX and EY, Equation 4 can be rewritten as the following:

$$F_X + F_Y = (-1)^{S_x} \times \begin{cases} 2^{1-B} \times \{0.X + 0.Y\} & Case\,0 : EX = EY = 0 \\ 2^{1-B} \times \{0.X + 1.Y \times 2^{EY-1}\} & Case\,1 : EX = 0 \,\&EY > 0 \\ 2^{1-B} \times \{2^{EX-1} \times 1.X + 0.Y\} & Case\,2 : EX > 0 \,\&EY = 0 \\ 2^{EX-B} \times \{1.X + 1.Y\} & Case\,3 : EX > 0 \,\&EY > 0 \,\&EX = EY \\ 2^{EX-B} \times \{1.X + 1.Y \times 2^{EY-EX}\} & Case\,4 : EX > 0 \,\&EY > 0 \,\&EX < EY \\ 2^{EY-B} \times \{2^{EX-EY} \times 1.X + 1.Y\} & Case\,5 : EX > 0 \,\&EY > 0 \,\&EX > EY \end{cases} \tag{6}$$

For Case 0, the number of distinct mantissa sums is only 1. For Case 1, the number of distinct mantissa sums is the same as the number of possible values of EY except 0, which is $2^n - 1$. Similarly, for Case 2, the number of distinct mantissa sums is also $2^n - 2$, but $0.X$+$1.Y$ has the same representation as $1.X$+$0.Y$ in case 1. For Case 3, the number of distinct mantissa sums is only 1. For Case 4, the number of distinct mantissa sums is the same as the number of possible values of $EY - EX$. Since both EX and EY can not be 0, the number of possible values of $EY - EX$ is $2^n - 2$. Therefore, the number of distinct mantissa sums is $2^n - 2$. Similarly, for Case 5, the number of distinct mantissa sums is also $2^n - 2$. Therefore the total number of distinct mantissa sums for the true addition is $2^{n+2} - 5$.

Similarly, Equation 5 can be rewritten as the following equation:

$$F_X + F_Y = (-1)^{S_x} \times \begin{cases} 2^{1-B} \times \{0.X - 0.Y\} & Case\,0: EX = EY = 0 \\ 2^{1-B} \times \{0.X - 1.Y \times 2^{EY-1}\} & Case\,1: EX = 0\,\&EY > 0 \\ 2^{1-B} \times \{2^{EX-1} \times 1.X - 0.Y\} & Case\,2: EX > 0\,\&EY = 0 \\ 2^{EX-B} \times \{1.X - 1.Y\} & Case\,3: EX > 0\,\&EY > 0\,\&EX = EY \\ 2^{EX-B} \times \{1.X - 1.Y \times 2^{EY-EX}\} & Case\,4: EX > 0\,\&EY > 0\,\&EX < EY \\ 2^{EY-B} \times \{2^{EX-EY} \times 1.X - 1.Y\} & Case\,5: EX > 0\,\&EY > 0\,\&EX > EY \end{cases} \quad (7)$$

For Case 5 and 4, the numbers of distinct mantissa sums are the same as those in the corresponding cases of true addition. For Case 3, the mantissa sum $1.X - 1.Y$ is the same as $0.X - 0.Y$ in Case 0. For both Case 1 and 2, the number of distinct mantissa sum is $2^n - 1$. Therefore, the number of of distinct mantissa sums for the true subtraction is also $2^{n+2} - 5$. Thus, the total number of distinct mantissa sums is $2^{n+3} - 10$. $\square$

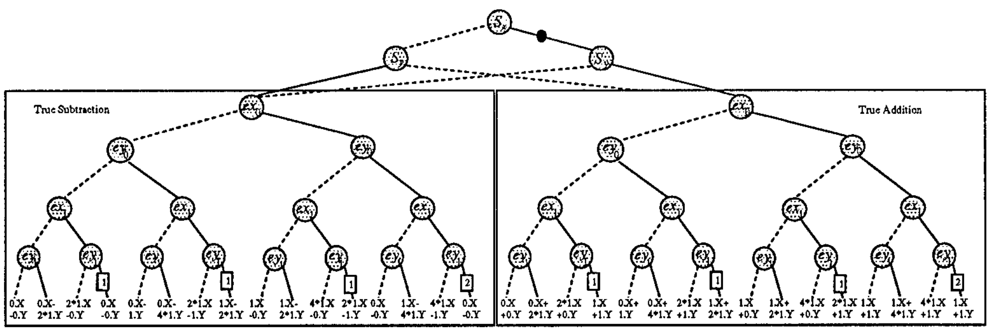


Figure 9: **Representations of the floating point addition.** For simplicity, the graph only shows sign bits, exponent bits and the possible combinations of mantissa sums.

Figure 9 illustrates the *PBHD representation of floating point addition with two exponent bits for each floating point operand. Observe that the negation edge reduces the graph size by half. There is no sharing among the sub-graphs for true addition and true subtraction. In true subtraction, $1.X - 1.Y$ has the same representation as $0.X - 0.Y$. Therefore, all $1.X - 1.Y$ entries are replaced by $0.X - 0.Y$. Since the number of distinct mantissa sums grows exponentially with the number of exponent bits, it can be shown that the total number of nodes grows exponentially with the size of exponent bits and grows linearly with the size of the mantissa part(See the appendix for the details of proofs). Floating point subtraction can be performed by the negation and addition operations. Therefore, it has the same complexity as addition.

In our experience, the sizes of the resulting graphs for multiplication and addition are hardly sensitive to the variables ordering of the exponent variables. They exhibit a linear growth for multiplication and exponential growth for addition for almost possible ordering of the exponent variables. It is more logical to put the variables with Shannon decompositions on the top of the variables with the other decompositions.

5 Comparisons with *BMD, HDD and K*BMD

The major difference between *PBHD and the other three diagrams is in their ability to represent functions that map Boolean variables into floating point values and their use of negation edges. Table 1 summarizes the differences between them.

Features	*PBHD	*BMD	HDD	K*BMD
Additive weight	No	No	No	Yes
Multiplicative weight	Powers of 2	GCD	No	GCD
Number of decompositions	3	1	6	3
Negation edge	Yes	No	No	No

Table 1: **Differences among four different diagrams.**

Compared to *BMDs [3], *PBHDs have three different decomposition types and a different method to represent and extract edge weights. These features enable *PBHDs to represent floating point functions effectively. *BMD's edge weights are extracted as the greatest common divisor (GCD) of two children. In order to verify the multiplier with a size larger than 32 bits, *BMDs have to use multiple precision representation for integers to avoid the machine's 32-bit limit. This multiple precision representation and the GCD computation are expensive for *BMDs in terms of CPU time. Our powers of 2 method not only allows us to represent the floating point functions but also improves the performance compared with *BMD's GCD method.

Compared with HDDs having six decompositions [6], *PBHDs have only three of them. In our experience, these three decompositions are sufficient to represent floating point functions and verify floating point arithmetic circuits. The other three decomposition types in HDDs may be useful for other application domains. Another difference is that *PBHDs have negation edges and edge weights, but HDDs do not. These features not only allow us to represent the floating point functions but also reduce the graph size.

*PBHDs have only multiplicative edge weights, but K*BMDs [9] allow additive and multiplicative weights at the same time. The method of extracting the multiplicative weights is also different in these two representations. *PBHDs extract the powers-of-2 and choose the minimum of two children, but K*BMDs extract the greatest common divisor of two children like *BMDs. The additive weight in K*BMDs can be distributed down to the leaf nodes in *PBHD by recursively distributing to one or two branches depending on the decomposition type of the node. In our experience, additive weights do not significantly improve the sharing in the circuits we verified. The sharing of the additive weight may occur in other application domains.

6 Experimental Results

We have implemented *PBHD with basic BDD functions and applied it to verify arithmetic circuits. Integer multiplier circuits and *BMD package can be obtained from Yirng-An Chen's WWW page[1]. The circuit structure for four different types of multipliers are manually encoded in a C program which calls the BDD and *BMD operations. We also integrated our *PBHD package with the C program. Our measurements are obtained on Sun Sparc 10 with 256 MB memory.

6.1 Integer Multipliers

Circuits		CPU Time (Sec.)			Memory(MB)		
		16	64	256	16	64	256
Add-Step	*BMD	1.40	15.38	354.38	0.67	0.77	1.12
	*PBHD	0.20	2.24	39.96	0.11	0.18	0.64
	Ratio	7.0	6.8	8.9	6.0	4.3	1.8
CSA	*BMD	1.61	26.91	591.70	0.67	0.80	2.09
	*PBHD	0.25	3.45	50.72	0.14	0.30	0.88
	Ratio	6.4	7.8	11.7	4.8	2.7	2.4
Booth	*BMD	2.05	34.09	782.20	0.70	0.86	1.84
	*PBHD	0.21	2.97	62.56	0.14	0.30	1.26
	Ratio	9.7	11.5	12.5	5.0	2.9	1.5
Bit-Pair	*BMD	1.21	17.35	378.64	0.70	0.86	2.34
	*PBHD	0.20	2.17	36.10	0.15	0.33	1.33
	Ratio	6.0	8.0	10.5	4.7	2.6	1.8

Table 2: **Performance comparison between *BMD and *PBHD for different integer multipliers.** Results are shown for three different words. The ratio is obtained by dividing the result of *BMD by that of *PBHD.

Table 2 shows the performance comparison between *BMD and *PBHD for different integer multipliers with different word sizes. *BMD results are close to the results published in the papers [3, 4]. For the CPU time, the complexity of *PBHDs for the multipliers still grows quadratically with the word size. Compared with *BMDs, *PBHDs are at least 6 times faster, since the edge weight manipulation of *PBHDs only requires integer addition and subtraction, but *BMDs require a multiple precision representation for integers and perform costly multiple precision multiplication and division. While increasing the word size, the *PBHD's speedup is increasing, because *BMDs requires more time to perform multiple precision multiplication and division operations. Interestingly, *PBHDs also use less memory than *BMDs, since the edge weights in *BMDs are explicitly represented by extra nodes, while *PBHDs embed edge weights into the node structure. The node sizes for both packages are 20 bytes.

[1]http://www.cs.cmu.edu/~yachen/home.html.

6.2 Floating Point Multipliers

Circuits	CPU Time (Sec.)			Memory(MB)		
	16	64	256	16	64	256
Add-Step	0.24	2.29	39.77	0.13	0.18	0.65
CSA	0.29	3.08	53.98	0.14	0.30	0.88
Booth	0.25	3.85	67.38	0.16	0.30	1.26
Bit-Pair	0.21	2.10	38.54	0.15	0.33	1.33

Table 3: **Performance for different floating point multipliers.** Results are shown for three different mantissa word size with fixed exponent size 11.

To perform floating point multiplication operations before the rounding stage, we introduced an adder to perform the exponent addition and logic to perform the sign operation in the C program. Table 3 shows CPU times and memory requirements for verifying the floating point multipliers with fixed exponent size 11. Observe that the complexity of verifying the floating point multiplier before rounding still grows quadratically. In addition, the computation time is very close to the time of verifying integer multipliers, since the verification time of an 11-bit adder and the composition and verification times of a floating point multiplier from integer mantissa multiplier and exponent adder are negligible. The memory requirement is also similar to that of the integer multiplier.

6.3 Floating Point Addition

	No. of Nodes		CPU Time (Sec.)		Memory(MB)	
Exponent Bits	23	52	23	52	23	52
4	4961	10877	0.23	0.68	0.37	0.74
5	10449	22861	0.67	1.30	0.68	1.06
6	21441	46845	1.13	3.50	1.13	1.99
7	434419	94829	2.65	6.87	1.90	3.79
8	**87457**	190813	**7.17**	16.82	**3.62**	7.46
9	175505	382797	14.98	41.34	7.15	14.75
10	351617	766781	33.35	103.17	14.26	29.47
11	703857	**1534765**	72.84	**262.35**	26.50	**54.89**
12	1408353	3070749	163.18	573.73	54.06	110.86
13	2817361	6142733	398.25	1303.84	112.49	226.00

Table 4: **Performance for floating point additions.** Results are shown for three different exponent word size with fixed mantissa size 23 and 52 bits.

Table 4 shows the performance measurements of floating point addition operations with different exponent bits and fixed mantissa sizes of 23 and 52 bits, respectively. Both the number of nodes and the required memory double, while adding one extra exponent bit. For the same exponent bit, the measurements for the 52-bit mantissa are approximately twice the corresponding measurements for the 23-bit mantissa. In other words, the complexity grows linearly with the mantissa's word size. Due to the cache behavior, the CPU time is not doubling, while increasing one extra exponent bit. For the double precision of IEEE standard 754 (the numbers of exponent and mantissa bits are 11 and 52 respectively), it only requires 54.89MB and 262.35 seconds. These values indicate the possibility of the verification of an entire floating point adder for IEEE double precision. For the IEEE extended precision, the floating point addition will require at least 226MB $\times$ 8 = 1808MB memory. In order to verify IEEE extended precision addition, it is necessary to avoid the exponential growth of floating point addition.

7 Conclusions and Future Works

We have described a new representation, *PBHD, to represent functions that map Boolean variables into integer or floating point values. We also applied *PBHDs to verify integer and floating point multipliers. For integer multipliers, the CPU time of *PBHDs grows quadratically and is at least 6 times faster than *BMDs. For the floating point multipliers, the verification time is close to the verification time of integer multipliers. In addition, we showed that the *PBHD representation for floating point multiplication grows linearly with the word size. For the floating point addition, we showed the graph size of the result grows exponentially with the word size of the exponent, but linearly with the size of the mantissa.

As mentioned in previous sections, we will further pursue handling infinite and NaN cases in the *PBHD representation. We need to develop some methodologies or introduce some special symbols to handle these cases. Another piece of future work is to handle the exponential growth of the floating point addition. Our representation for the floating point addition represents the precise values of all possible combinations, but in the actual circuit design, there are only about 200 interesting mantissa sums. Based on this knowledge, we will develop the methodology to avoid the exponential growth of floating point addition.

To verify circuit designs automatically, we would like to integrate *PBHD package into word-level SMV [7] and extend the word-level SMV if needed. Another possible approach is to integrate our *PBHD package into the ACV system [4] and extend the ACV system to handle the floating point arithmetic circuits.

After integrating our *PBHD package into word-level SMV or ACV, our next step is to look into the rounding stage and the entire floating-point multipliers and adders. Earlier results using HDDs [5] show that the rounding stage itself can be handled. Since *PBHD is more powerful than HDDs, we believe that the rounding stage can be verified using *PBHDs. We also believe that it is possible to verify floating point multipliers using *PBHDs. To verify the entire floating point adders, we need to develop some methodologies to avoid the exponential growth.

Appendix: Complexity Proofs of Floating Point Addition

In this section, we prove that the exact graph size of floating point addition under a fixed variable ordering grows exponentially with the size of the exponent and linearly with the size of the mantissa.

Assume that the sizes of the exponent and the mantissa are n and m bits, respectively. We assume that the variable ordering is S_x, S_y, cx_0, cy_0, ..., cx_{n-1}, cy_{n-1}, x_{m-1}, ..., x_0, y_{m-1}, ..., y_0.

Lemma 1 *The size of the mantissa part of the resulting graph is* $2^{n+1}(7m-1) - 20m - 4$, *where* n *and* m *are the numbers of bits of the exponent and mantissa parts respectively.*

Proof: Theorem 2 showed that the number of distinct mantissa sums is $2^{n+3} - 10$. Except the leaf nodes, each mantissa sum can be represented by $2m$ nodes, but there is some sharing among the mantissa graphs. First, let us look at the sharing among the mantissa sums of true addition. For case 4 in Equation 6, the graphs to represent function $0.X + 2^{EY-1} \times 1.Y$ share the same subgraph $1.Y$, which is also in the graph representing function $1.X + 0.Y$. Thus, there are $2^n - 1$ distinct mantissa sums to share the same graph($1.Y$). Again, the graphs to represent $1.X + 1.Y$ in case 2 and $2 \times 1.X + 0.Y$ in case 3, share the sub-graph $2 + 0.Y$, since $1.X + 1.Y = 0.X + (2 + 0.Y)$ and $2 \times 1.X + 0.Y = 2 \times 0.X + (2 + 0.Y)$. Therefore, we have to subtract $(2^n - 1)m$ nodes from the total nodes.

Then, let us look at the true subtraction. First, the graph to represent $0.X - 0.Y$ shares the sub-graph $0.Y$ with $0.X + 0.Y$ in true addition, because of the negation edge. For Case 4 in Equation 7, the graphs to represent function $0.X - 2^{EY-1} \times 1.Y$ share the same subgraph $1.Y$ in true addition. The graphs to represent $1.X - 0.Y$ in case 3 and $2 \times 1.X - 1.Y$ in case 0, share the sub-graph $1 - 0.Y$, since $1.X + 1.Y = 0.X + (1 - 0.Y)$ and $2 \times 1.X - 1.Y = 2 \times 0.X + (1 - 0.Y)$. Therefore, we have to subtract $(2^n + 1)m$ nodes from the total nodes. Thus, the number of non-leaf nodes to represent these distinct mantissa sums is $(7 \times 2^n - 10) \times (2m)$.

The leaf nodes 1 and 0 are referenced by these non-leaf nodes. For true addition, the number of leaf nodes, except leaf node 1, is $2^n - 2$, since the leaf nodes of the mantissa sum for $EX < EY$ can be shared with the mantissa sum for $EX > EY$. To be specific, the leaf nodes are generated by the sum of the leading 1s in the form of $1 + 1 \times 2^{EY-EX}$ or $1 \times 2^{EX-EY} + 1$, and there are only $2^n - 2$ sums. Similarly, for true subtraction, there are $2^n - 4$ leaf nodes, but the leaf nodes 3 $(2^2 - 1)$ and 0 $(2^0 - 1)$ already exist. Thus, the total number of leaf nodes is $2 + (2^n - 2) + (2^n - 4) = 2^{n+1} - 4$. Therefore, the size of the mantissa part of resulting graph is $(7 \times 2^n - 10) \times (2m) + 2^{n+1} - 4 = 2^{n+1}(7m - 1) - 20m - 4$. $\square$

Lemma 2 *For all* $n \geq 2$, *the number of *PBHD nodes of the exponent part of the resulting graph is* $5 \times 2^{n+2} - 16 \times n - 18$.

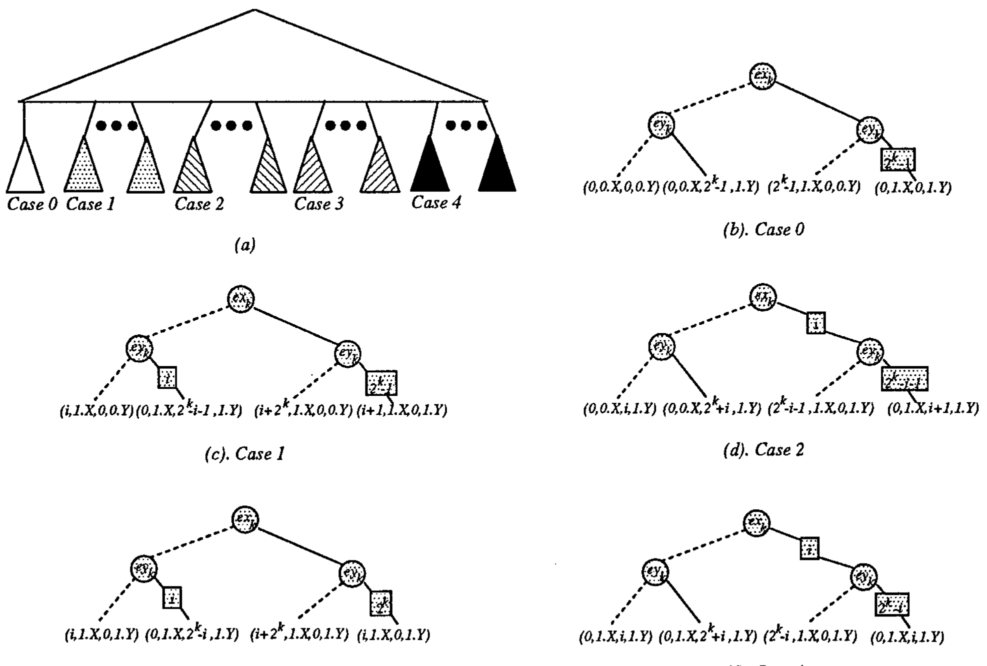
Proof: As mentioned before, the resulting graph can be divided into two parts: true addition and true subtraction. First, we prove that the number of nodes of the exponent part for true addition is $5 \times 2^{n+1} - 8 \times n - 9$. We prove this claim by the induction on the number of exponent bits n.

Base Case: If $n = 2$, the number of exponent nodes for true addition is $5 \times 2^{2+1} - 8 \times 2 - 9 = 15$ as shown in Figure 9.

Induction Step: Assume the claim holds for $n = k$. To prove that the claim holds for $n = k+1$, let EX_k and EY_k represent the low k bits of EX and EY. Thus, EX is represented as $2^k \times cx_k + EX_k$. Based on the values of EX_k and EY_k, Equation 4 can be rewritten as the following:

$$F_X + F_Y = (-1)^{S_x} \times \begin{cases} 2^{1-B} \times \{2^{((2^k-1)\times ex_k)} \times G + 2^{((2^k-1)\times ey_k)} \times H\} & Case\ 0: EX_k = EY_k = 0 \\ 2^{1-B} \times \{2^{(EX_k-1+(2^k-1)\times ex_k)} \times 1.X + H \times 2^{((2^k-1)\times ey_k)}\} & Case\ 1: EX_k > 0\ \&EY_k = 0 \\ 2^{1-B} \times \{G \times 2^{((2^k-1)\times ex_k)} + 1.Y \times 2^{(EY_k-1+(2^k-1)\times ey_k)}\} & Case\ 2: EX_k = 0\ \&EY_k > 0 \\ 2^{EY_k-B} \times \{(2^{EX_k-EY_k+2^k\times ex_k)} \times 1.X + 1.Y \times 2^{(2^k\times ey_k)}\} & Case\ 3: EX_k > 0\ \&EY_k > 0\ \&EX_k > EY_k \\ 2^{EX_k-B} \times \{2^{(2^k\times ex_k)} \times 1.X \times +1.Y \times 2^{(EY_k-EX_k+2^k\times ey_k)}\} & Case\ 4: EX_k > 0\ \&EY_k > 0\ \&EX_k \le EY_k \end{cases} \tag{8}$$

where G (H) is $0.X$ ($0.Y$) if cx_k (cy_k) is 0; otherwise, G (H) is $1.X$ ($1.Y$). Figure 10.a illustrates the distinct sub-graphs after expanding variable cy_{k-1}. These sub-graphs are divided into five types, according to the cases in Equation 8. For Case 0, there is only one distinct sub-graph. For Case 1, there are $2^k - 1$ distinct sub-graphs, since the number of possible value of EX_k is $2^k - 1$ and each value of EX_k will generate a unique function. Similarly, there are $2^k - 1$, $2^k - 2$, and $2^k - 1$ distinct sub-graphs for Case 2, 3, and 4, respectively. Thus, the total number of distinct sub-graphs is $2^{k+2} - 4$.



Case 0
Case 1
Case 2
Case 3
Case 4
(a)
ex_k
ey_k
2^k-1
(0,0.X,0,0.Y) (0,0.X,2^k-1,1.Y) (2^k-1,1.X,0,0.Y) (0,1.X,0,1.Y)
(b). Case 0
i
(i,1.X,0,0.Y) (0,1.X,2^k-i-1,1.Y) (i+2^k,1.X,0,0.Y) (i+1,1.X,0,1.Y)
(c). Case 1
2^k-i-1
(0,0.X,i,1.Y) (0,0.X,2^k+i,1.Y) (2^k-i-1,1.X,0,1.Y) (0,1.X,i+1,1.Y)
(d). Case 2
2^k
(i,1.X,0,1.Y) (0,1.X,2^k-i,1.Y) (i+2^k,1.X,0,1.Y) (i,1.X,0,1.Y)
(e). Case 3
2^k-i
(0,1.X,i,1.Y) (0,1.X,2^k+i,1.Y) (2^k-i,1.X,0,1.Y) (0,1.X,i,1.Y)
(f). Case 4

Figure 10: **Distinct sub-graphs after variable** cy_{k-1}. (a) Distinct sub-graphs after variable cy_{k-1} are divided into 5 types shown in graphs (b) to (f) which serve as template with a parameter i. (b) Case 0 only has one distinct graph. (c) $0 \le i = EX_k - 1 \le 2^k - 2$. (d) $0 \le i = EY_k - 1 \le 2^k - 2$. (e) $1 \le i = EX_k - EY_k \le 2^k - 2$. (f) $1 \le i = EY_k - EX_k \le 2^k - 2$.

Figures 10.b shows the sub-graph for Case 0. In the graphs, each tuple (i, P, j, Q) represents $2^i \times P + 2^j \times Q$. For example, tuple $(0, 0.X, 0, 0.Y)$ represents $2^0 \times 0.X + 2^0 \times 0.Y$. Figures 10.c to Figures 10.f show the graphs with a parameter i for Cases 1, 2, 3 and 4, which serve as the template of the graphs in the cases. For instance, the graph in Case 1 with $i = 1$ represents the function $2^{(EX_k-1+(2^k-1)\times ex_k)} \times 1.X + II \times 2^{((2^k-1)\times ey_k)}$ with $EX_k = 1$ in Case 2 of Equation 8.

Since each sub-graph is distinct, the nodes with variable cx_k are unique (i.e. no sharing). Observing from these five types of sub-graphs, the possible sharing among the nodes with variable cy_k is these cases: the cy_k nodes in case 2 share with that in cases 3 and 4, and the nodes in case 3 share with that in case 4. For the first case, the possible sharing is the right cy_k nodes in Figure 10.e and Figure 10.g. Observe that these two cy_k node will be that same in the graph with $i = j$ in case 2 and the graph with $i = j + 1$ in case 4. Since the possible values of i are ranged from 0 to $2^k - 3$, there are $2^k - 2$ cy_k nodes shared. When $i = 2^k - 2$, the right cy_k node in the graph of case 2 will be shared with the left cy_k node in the graph with i=1 in Figure 10.e. Therefore, all of the right cy_k nodes in Case 2 are shared nodes and are $2^k - 1$ nodes. For the second case, the possible sharing is the left cy_k node in Figure 10.e and the right cy_k node in Figure 10.f. Observe that when $i_1 + i_2 = 2^k$, the left cy_k node in the graph with $i = i_1$ in case 3 is the same as the right cy_k node in the graph with $i = i_2$ in case 4. Since $2 \leq i_1 \leq 2^k - 2$ and $0 \leq i_2 \leq 2^k - 2$, there are $2^k - 3$ nodes shared. Therefore, the total number of exponent nodes are $5 \times 2^{k+1} - 8 \times k - 9 + 3 \times (2^{k+2} - 4) - (2^k - 1) - (2^k - 3) = 5 \times 2^{(k+1)+1} - 8 \times (k+1) - 9 = 5 \times 2^{n+1} - 8 \times n - 9$.

Similarly, the number of nodes of the exponent part for true subtraction is $5 \times 2^{n+1} - 8 \times n - 9$. Therefore, the size of the exponent part of the resulting graph is $5 \times 2^{n+2} - 16 \times n - 18$. □

Theorem 3 *For the floating-point addition, the size of the resulting graph is* $2^{n+1} \times (7m + 9) - 20m - 16n - 19$.

Proof: The size of the resulting graph is the sum of the nodes for the sign, exponent and mantissa parts. The nodes for the sign part are 3 as shown in Figure 9. Lemma 1 and 2 have shown the sizes of the mantissa and exponent parts respectively. Therefore, their sum is $2^{n+1} \times (7m + 9) - 20m - 16n - 19$. □

References

[1] BRACE, K., RUDELL, R., AND BRYANT, R. E. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference* (June 1990), pp. 40–45.

[2] BRYANT, R. E., AND CHEN, Y.-A. Verification of arithmetic functions with binary moment diagrams. Tech. Rep. CMU-CS-94-160, School of Computer Science, Carnegie Mellon University, 1994.

[3] BRYANT, R. E., AND CHEN, Y.-A. Verification of arithmetic circuits with binary moment diagrams. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference* (1995), pp. 535–541.

[4] CHEN, Y.-A., AND BRYANT, R. E. ACV: An arithmetic circuit verifier. In *Proceedings of the International Conference on Computer-Aided Design* (November 1996), pp. 361–365.

[5] CHEN, Y.-A., CLARKE, E. M., HO, P.-H., HOSKOTE, Y., KAM, T., KHAIRA, M., O'LEARY, J., AND ZHAO, X. Verification of all circuits in a floating-point unit using word-level model checking. In *Proceedings of the Formal Methods on Computer-Aided Design* (November 1996), pp. 19–33.

[6] CLARKE, E. M., FUJITA, M., AND ZHAO, X. Hybrid decision diagrams overcoming the limitations of MTBDDs and BMDs. In *Proceedings of the International Conference on Computer-Aided Design* (November 1995), pp. 159–163.

[7] CLARKE, E. M., KHAIRA, M., AND ZHAO, X. Word level model checking – Avoiding the Pentium FDIV error. In *Proceedings of the 33rd ACM/IEEE Design Automation Conference* (June 1996), pp. 645–648.

[8] CLARKE, E. M., MCMILLAN, K., ZHAO, X., FUJITA, M., AND YANG, J. Spectral transforms for large boolean functions with applications to technology mapping. In *Proceedings of the 30th ACM/IEEE Design Automation Conference* (June 1993), pp. 54–60.

[9] DRECHSLER, R., BECKER, B., AND RUPPERTZ, S. K*BMDs: a new data struction for verification. In *Proceedings of European Design and Test Conference* (March 1996), pp. 2–8.

[10] DRECHSLER, R., SARABI, A., THEOBALD, M., BECKER, B., AND PERKOWSKI, M. A. Efficient representation and manipulation of switching functions based on ordered Kronecker functional decision diagrams. In *Proceedings of the 31st ACM/IEEE Design Automation Conference* (June 1994), pp. 415–419.